UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1269

Programme 3
Réseaux et Systèmes Répartis

SYNCHRONOUS DISTRIBUTED ALGORITHMS : A PROOF SYSTEM

Michel ADAM
Jean-Michel HELARY

Juillet 1990

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

Synchronous distributed algorithms: a proof system

Algorithmes Distribués Synchrones: un système de preuve

Michel Adam* Jean-Michel Hélary†

Equipe Algorithmes Distribués et Protocoles

Publication Interne n° 538 - Juin 1990 - 20 Pages

Abstract

Simulating synchronism on asynchronous networks allows to design synchronous distributed algorithms. This design relies upon a language with a clear operationnal semantic and an assertional proof system, consistent and complete. As an illustration, a synchronous election algorithm on an arbitrary topology network is formally derived and proved.

Résumé

Simuler le synchronisme sur les réseaux asynchrones permet de concevoir des algorithmes distribués synchrones. Cette conception repose sur une sémantique opérationnelle clairement définie et un système de preuve par assertions, correct et complet. A titre d'exemple, un algorithme synchrone d'élection sur un réseau à topologie quelconque est systématiquement dérivé et prouvé.

1 Introduction

One of fundamental problems in distributed programming results from impossibility to capture instantaneous global states. This problem is mainly due to arbitrary propagation message delay on channels. To overcome this difficulty, essentially three classes of solutions have been proposed :

1. Building a virtual global time consistent with causality relation between events [Lam78, MO83, Fid88, Mat88]
2. Building a virtual central memory by repeated computation of coherent global states [CL85, LY87, Mat88, HPR89].
3. Building a virtual synchronous network, that is to say a network in which an upper bound for transmission and processing delays of any message is known [Awe85, HR88]

*ENSTBr, antenne de Rennes, rue de la chataigneraie, BP 78, 35512 Cesson-Sévigné Cedex, France
†IRISA, Campus de Beaulieu, 34042 Rennes Cedex, France
‡This work was supported by the French Research Program C^3 on Parallelism and Distributed Computing

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(U.R.A. 227)
UNIVERSITE DE RENNES I I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(UNITE DE RECHERCHE DE RENNES)

As can be seen, in each of these approaches a virtual machine is provided, which allows to design distributed programs within an easier context. This paper relates to the third solution.

Synchronous networks support the design of so-called *distributed synchronous programs (SDP)*. This concept was introduced by B.Awerbuch [Awe85], which proposed the concept of *synchronizer*: the latter allow SDP to be run on arbitrary asynchronous distributed networks (interpretor of SDP on asynchronous network). Later, synchronizers for different kinds of networks were designed [PU87, CCGZ87, FLS87] and a comparative study based upon experimentation can be found in [AIR88].

Design and control of SDP is easier than for general distributed programs. Thanks to synchronism, it is possible to conceive SDP as a sequence of synchronous steps, or *pulses*, such that each pulse consists for each process of an emission phase followed by a reception-processing phase and *each received message has been sent during the same pulse.* This last property does capture the essentiality of synchronism hypothesis. Taking into account this kind of global sequentiality leads to extend to SDP formal methods suited for a centralized sequential context. Our paper focuses on definition and use of a language and of a proof system for SDP.

In the second section, we define a language for SDPs, whose syntax is similar to CSP's [Hoa78]. In the third section, an operational semantic for this language is given, in a formalism similar to Hennessy and Plotkin's [Hen79]. The fourth section is devoted to a proof system for our SDP language, which is illustrated in the fifth section where a distributed synchronous election algorithm on an arbitrary topology is obtained and proved by derivation steps *à la* Gries [Gri81].

Some proof systems have previously been proposed in parallel contexts, e.g. [OG76] for shared-memory parallel contexts, [AFP80, Sou84] for CSP, etc. To our knowledge, the proof system presented here is the first publisheed devoted to SDP.

Some results and proofs which are not detailed here can be found in [Ada90]. Let's recall some general notations and concepts used throughout of the paper :

The synchronous network supporting SDP is represented by a graph $G = (V, E)$ where $V = \{P_1, P_2, \ldots, P_n\}$ is the set of nodes, E is the set of channels; each node supports a single process with its own local context; two processes are *neighbours* if they are linked by a channel in G. Processes share no memory and only neighbours can communicate by exchanging messages through their connecting channel.

The specification of a SDP consists of :

- initializing the variables,
- specifying messages to be sent during emission phase,
- specifying actions to perform upon reception of messages during reception phase,
- specifying termination condition.

2 A language for synchronous distributed algorithms

In this section we propose a language for synchronous distributed algorithms expression. The syntax is similar to CSP's, as defined by Dijkstra [Dij76] and Hoare [Hoa78]. Apart from the classical constructions, a new one is added, to take synchronism into account: the *pulsation loop*.

Locally on each process, the following constructions are avalaible (informal semantic is given; see next section for an operationnal semantic):

- Null action : **skip**
- Assignment : $\mathbf{x := e}$
- Sequentiality : $\mathbf{S1; S2}$
- Non determinist choice : $[\Box_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{b_j} \rightarrow \mathbf{S_j}]$
  Action $\mathbf{S_j}$ is performed only if the corresponding guard b_j is true. If several guards are true, only one corresponding action is performed (choice is non determinist). At least one guard must be true.
- Local loop : $*[\Box_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{b_j} \rightarrow \mathbf{S_j}]$
  If all guards $\mathbf{b_j}$ are false, equivalent to **skip**, otherwise equivalent to the sequence $\mathbf{S_l}; *[\Box_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{b_j} \rightarrow \mathbf{S_j}]$ where l is such that b_l is true.
- Emission of a message : $\mathbf{P_j!e}$
  The process sends the value e to process P_j
- Reception of a message : $[\mathbf{P_j?x} \rightarrow \mathbf{S}]$
  If P_j sends a message, the value contained in the message is assigned to x, then **S** is performed. Otherwise, **skip**
- Pulse loop : $\otimes[\mathbf{b_i} \rightarrow \mathbf{S_i}]$
  At the beginning of a pulse, if b_i is true then $\mathbf{S_i}$ is performed. Otherwise, the synchronous process P_i is terminated. It is this last construction which takes into account the synchronism.

The following abbreviations will be used to simplify the text of programs :

- $[\nabla_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{P_j!e}]$ means $\mathbf{P_1!e; P_2!e; \cdots; P_{m-1}!e; P_m!e}$
- $[\nabla_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{P_j?x} \rightarrow \mathbf{a}]$ means $[\mathbf{P_1?x} \rightarrow \mathbf{a}]; [\mathbf{P_2?x} \rightarrow \mathbf{a}]; \cdots; [\mathbf{P_{m-1}?x} \rightarrow \mathbf{a}]; [\mathbf{P_m?x} \rightarrow \mathbf{a}]$

A synchronous distributed algorithm will be denoted $[\mathbf{S_1} \parallel \cdots \parallel \mathbf{S_i} \parallel \cdots \parallel \mathbf{S_n}]$, where $\mathbf{S_i}$ is the text of the sequential process P_i. Each of the $\mathbf{S_i}$ is of the form $\mathbf{A_i}; \otimes[\mathbf{b_i} \rightarrow \mathbf{S_i}]$. $\mathbf{A_i}$ is the *initialization* part of $\mathbf{S_i}$, involving only variable assignments (communications are prohibited in this part). Boolean expression b_i is the *guard of pulse loop*. $\mathbf{S_i}$ is the *pulse loop body*. It is made of two sequential parts: $\mathbf{S_i \equiv EM_i; REC_i}$, where

- $\mathbf{EM_i}$ is the *emission part*; the only enabled communications are emissions, and at most one message per outgoing channel can be sent during a pulse,
- $\mathbf{REC_i}$ is the *reception part*; the only enabled communications are receptions, and at most one message per ingoing channel can be received during a pulse.

The operationnal semantic, described below, insures these constraints. It works like a language interpretor; should one constraint be violated, the system would block.

2.1 An operationnal semantic

To give a language its meaning requires to provide its semantic. Moreover, an *operationnal* semantic is sufficient to establish correctness and completness of proof systems. An operationnal semantic is a *language interpretor*: the state of a distributed program must be defined, and for each language construction, the transition to the next state must be stated.

The semantic used here is given in a form similar to Hennessy and Plotkin ([Hen79]). To fit the synchronous model, we define the state of the distributed program as the collection of states of each process and of each channel:

$$< \{S_1, \sigma_1\}, \{S_2, \sigma_2\}, \cdots, \{S_{n-1}, \sigma_{n-1}\}, \{S_n, \sigma_n\}, C, R >$$

where:

- The state of a process P_i is the 2-uple $\{S_i, \sigma_i\}$ with S_i : sequence of actions remaining to execute and σ_i : state of P_i's local context (memory, etc., except adjacent channels). Actually, σ_i is a function defined over Var_i, the set of P_i's local variables, into a set N of values, e.g. natural integers. More precisely, the state of P_i will be denoted by $< S_i, \sigma_i >$ during the emission phase and by $\ll S_i, \sigma_i \gg$ during the reception phase. $\{S_i, \sigma_i\}$ denotes either $< \ldots >$ or $\ll \ldots \gg$.
- The state of channels is made of two functions $C : E \longrightarrow N \cup Nil$ and $R : E \longrightarrow \{\mathbf{true}, \mathbf{false}\}$. Recall that in our synchronous model, a channel has at most one message per pulse, and at any time between its emission and the end of the pulse it can be read or not yet read. In consequence, the value of $C(P_i, P_j)$, denoted C_{ij}, is equal to Nil if the channel is empty, or to the value of the message in the channel otherwise. Also, the value of $R(P_i, P_j)$, denoted R_{ij}, equals **true** iff the message contained in the channel has been read by the receptor P_j.

Classically,

$$\begin{array}{c} < \{S_1, \sigma_1\}, \{S_2, \sigma_2\}, \cdots, \{S_n, \sigma_n\}, \{S_n, \sigma_n\}, C, R > \\ \downarrow \\ < \{S'_1, \sigma'_1\}, \{S'_2, \sigma'_2\}, \cdots, \{S'_{n-1}, \sigma'_{n-1}\}, \{S'_n, \sigma'_n\}, C', R' > \end{array}$$

denotes a non-deterministic transition from the first state to the second one, during one step of the computation.

When an assertion p is true with respect to the processes and channels contexts $(\sigma_1, \cdots, \sigma_n, C, R)$, it is denoted by $\models p(\sigma_1, \cdots, \sigma_n, C, R)$. The abbreviation $\models p(\sigma_i)$ means that p is restricted to local constants and variables belonging to Var_i. Finally, $\sigma_i(e)$ denotes the value in N of expression e in the context σ_i, and $\sigma_i[e/x]$ means that context σ_i is changed into a new one where all variables but x remain unchanged, and x takes the value e. We use the following abbreviation : Nil instead of C means that all the channels are empty, and $False$ (resp. $True$) instead of R means that no (resp every) channel has been read.

2.2 An overview of the rules

E denotes the empty sequence. It is semantically equivalent to execute $E; S$, $S; E$ or S. We don't give the rules corresponding to the local internal instructions *skip, assignment, choice, loop, sequentiality* which are classical. We restrict ourselves to the constructions specific to our synchronous model : pulse loop, pulse sequentiality, communication, passage from emission phase to reception phase, termination. The complete set of rules can be found in [Ada90]

- Pulse loop

$$\frac{\models b_i(\sigma_i)}{\begin{array}{c}< \{S_1,\sigma_1\},\cdots,< \otimes[b_i \rightarrow S_i],\sigma_i >,\cdots,\{S_n,\sigma_n\},C,R > \\ \downarrow \\ < \{S_1,\sigma_1\},\cdots,< S_i \circ \otimes[b_i \rightarrow S_i],\sigma_i >,\cdots,\{S_n,\sigma_n\},C,R >\end{array}}$$

If the guard is true, the pulse body will be executed. State remains the same. The pulse loop will be reexecuted at the beginning of next pulse. The behaviour is similar to that of a local loop.

$$\frac{\models \neg b_i(\sigma_i)}{\begin{array}{c}< \{S_1,\sigma_1\},\cdots,< \otimes[b_i \rightarrow S_i],\sigma_i >,\cdots,\{S_n,\sigma_n\},C,R > \\ \downarrow \\ < \{S_1,\sigma_1\},\cdots,\ll E \circ \otimes[b_i \rightarrow S_i],\sigma_i \gg,\cdots,\{S_n,\sigma_n\},C,R >\end{array}}$$

If, on the contrary, the guard is false, no action will be performed during the pulse. State remains unchanged. Since proceses don't share memory, the guard will remain false for ever: P_i as terminated. However, the guard will be reevaluated at the beginning of next pulse, since the synchronous program will terminate only when all the processors will have terminated. Empty sequence E enables the pulse synchronization, denoted here by $\circ$.

- Pulse sequentiality

$$\frac{\models \bigwedge_{i=1}^{n}(\neg b_i \wedge (\bigwedge_{P_j \in V_i} C_{ij} = nil \vee R_{ij}))(\sigma_1,\cdots,\sigma_n,C,R)}{\begin{array}{c}<\ll E \circ \otimes[b_1 \rightarrow S_1],\sigma_1 \gg,\cdots,\ll E \circ \otimes[b_n \rightarrow S_n],\sigma_n \gg,C,R > \\ \downarrow \\ <\ll E,\sigma_1 \gg,\cdots,\ll E,\sigma_n \gg,Nil,False >\end{array}}$$

Channels are empty or read, all pulse guards are false, all processes have ended their pulse : final state is reached.

$$\frac{\models \bigvee_{i=1}^{n}(b_i \wedge (\bigwedge_{P_j \in V_i} C_{ij} = nil \vee R_{ij}))(\sigma_1,\cdots,\sigma_n,C,R)}{\begin{array}{c}<\ll E \circ \otimes[b_1 \rightarrow S_1],\sigma_1 \gg,\cdots,\ll E \circ \otimes[b_n \rightarrow S_n],\sigma_n \gg,C,R > \\ \downarrow \\ << \otimes[b_1 \rightarrow S_1],\sigma_1 >,\cdots,< \otimes[b_n \rightarrow S_n],\sigma_n >,Nil,False >\end{array}}$$

Channels are empty or read, at least one guard is true, all processors have ended their pulse : pulse loop must be executed until all processes terminate. Processes pass from the reception state $\ll \ldots \gg$ to the emission state $< \ldots >$.

Note that in both cases, channels are empty at the beginning of a pulse.

- Emission of a message.

$$\frac{\models (C_{ij} = nil)(\sigma_1, \cdots, \sigma_n, C, R)}{\begin{array}{c} < \{S_1, \sigma_1\}, \cdots, < P_j!e, \sigma_i >, \cdots, \{S_n, \sigma_n\}, C, R > \\ \downarrow \\ < \{S_1, \sigma_1\}, \cdots, < E, \sigma_i >, \cdots, \{S_n, \sigma_n\}, C[\sigma_i(e)/C_{ij}], R > \end{array}}$$

Emission is enabled since channel is empty: if a process sends more than one message on the same channel during a pulse, it blocks. Note that this rule is similar to the assignment of value e to the channel "variable" C_{ij}. Other contexts remain unchanged.

- Passage from $< \ldots >$ to $\ll \ldots \gg$

$$\begin{array}{c} < \{S_1, \sigma_1\}, \cdots, < [P_j?x \rightarrow S], \sigma_i > \cdots, \{S_n, \sigma_n\}, C, R > \\ \downarrow \\ < \{S_1, \sigma_1\}, \cdots, \ll [P_j?x \rightarrow S], \sigma_i \gg, \cdots, \{S_n, \sigma_n\}, C, R > \end{array}$$

P_i checks an ingoing channel. Thus its emission phase is terminated and it goes into its reception phase.

$$\frac{\begin{array}{c} < \{S_1, \sigma_1\}, \cdots, < \otimes[b_i \rightarrow S_i], \sigma_i >, \cdots, \{S_n, \sigma_n\}, C, R > \\ \downarrow \\ < \{S'1, \sigma'_1\}, \cdots, < E \circ \otimes[b_i \rightarrow S_i], \sigma'_i >, \cdots, \{S'_n, \sigma'_n\}, C', R' > \end{array}}{\begin{array}{c} < \{S_1, \sigma_1\}, \cdots, < \otimes[b_i \rightarrow S_i], \sigma_i >, \cdots, \{S_n, \sigma_n\}, C, R > \\ \downarrow \\ < \{S'_1, \sigma'_1\}, \cdots, \ll E \circ \otimes[b_i \rightarrow S_i], \sigma'_i \gg, \cdots, \{S'_n, \sigma'_n\}, C', R' > \end{array}}$$

P_i has terminated a pulse without checking ingoing channels. Thus its emission phase is terminated. It goes into its (empty) reception phase. This is necessary to let other processes read their received messages (if any).

- Reception of a message. All processes are in their reception phase, in particular all messages belonging to the present pulse have been sent. P_i can check its ingoing channels.

$$\frac{\models (C_{ji} \neq Nil \wedge \neg R_{ji})(\sigma_1, \cdots, \sigma_n, C, R)}{\begin{array}{c} < \ll S_1, \sigma_1 \gg, \cdots, \ll [P_j?x \rightarrow S], \sigma_i \gg, \cdots, \ll S_n, \sigma_n \gg, C, R > \\ \downarrow \\ < \ll S_1, \sigma_1 \gg, \cdots, \ll S, \sigma_i[C_{ji}/x] \gg, \cdots, \ll S_n, \sigma_n \gg, C, R[true/R_{ji}] > \end{array}}$$

Channel (P_j, P_i) contains an unread message, whose value is assigned to x, then sequence S associated with this reception will be performed. R_{ji} is assigned the value **true**, so that P_i will block if it tries to read another value on that channel.

$$\frac{\models C_{ji} = Nil(\sigma_1, \cdots, \sigma_n, C, R)}{\begin{array}{c} <\ll S_1, \sigma_1 \gg, \cdots, \ll [P_j?x \rightarrow S], \sigma_i \gg, \cdots, \ll S_n, \sigma_n \gg, C, R > \\ \downarrow \\ <\ll S_1, \sigma_1 \gg, \cdots, \ll E, \sigma_i \gg, \cdots, \ll S_n, \sigma_n \gg, C, R > \end{array}}$$

Channel (P_j, P_i) is empty: there is no action to perform.

- Termination

$$\begin{array}{c} << S_1, \sigma_1 >, \cdots, < S_n, \sigma_n >, Nil, False > \\ \downarrow^* \\ <\ll E, \sigma'_1 \gg, \cdots, \ll E, \sigma'_n \gg, Nil, False > \end{array}$$

$\downarrow^*$ is the transitive closure of $\downarrow$. Starting from an initial state with empty channels and in emission state, processes reach a state where no action is to execute and channels are empty.

Now we give some definitions :

Definition 2.1 (Semantic) *$\mathcal{M}$ is a function which, to a distributed synchronous program on $G = (V, E)$ with $|V| = n$ and to a state associates a state.*

$$\begin{array}{l} \mathcal{M}[S_1 \parallel \cdots \parallel S_n](\sigma_1, \cdots, \sigma_n, C, R) = (\sigma'_1, \cdots, \sigma'_n, C', R') \\ \qquad\qquad \textit{if and only if} \\ \qquad < \{S_1, \sigma_1\}, \cdots, \{S_n, \sigma_n\}, C, R > \\ \qquad\qquad \downarrow^* \\ \qquad < \{E, \sigma'_1\}, \cdots, \{E, \sigma'_n\}, C', R' > \end{array}$$

Definition 2.2

$$\begin{array}{c} \models \{p\}[S_1 \parallel \cdots \parallel S_n]\{q\} \\ \textit{if and only if} \\ \forall(\sigma_1, \cdots, \sigma_n, C, R), \\ \models p(\sigma_1, \cdots, \sigma_n, C, R) \Rightarrow q(\mathcal{M}[S_1 \parallel \cdots \parallel S_n](\sigma_1, \cdots, \sigma_n, C, R)) \end{array}$$

A "local" version of this definition can be stated :

Definition 2.3

$$\begin{array}{c} \models \{p\}S_i\{q\} \\ \textit{if and only if} \\ \forall(\sigma_1, \cdots, \sigma_n, C, R), \\ \models p(\sigma_1, \cdots, \sigma_n, C, R) \Rightarrow q(\mathcal{M}[E \parallel \cdots \parallel S_i \parallel \cdots \parallel E](\sigma_1, \cdots, \sigma_n, C, R)) \end{array}$$

3 The proof system

3.1 Projection of an assertion

In a distributed context, some assertions may involve variables belonging to different processes. Such assertions will have to be handled, like purely local ones, by any proof system, whence the need to state a decomposition concept, based upon the notion of *projection* of an assertion p onto a process P_i, denoted $proj_i(p)$. Let $Var(p)$ be the set of variables involved in assertion p, and $Var = \bigcup_{i=1}^{n} Var_i$ the set of all variables of all processes. The projection $proj_i(p)$ must satisfy two properties:

1. $\forall v \in Var(proj_i(p)),\ v \in Var \Rightarrow v \in Var_i$

2. $\bigwedge_{i=1}^{n} proj_i(p) \Rightarrow p$

(1) means that $proj_i(p)$ may involve some "new" variables which belong to no processes. They are in fact *auxiliary variables*.

(2) is an "orthogonality" property, meaning that an assertion can be restored from the set of its projections over all processes.

Let's first show on a small example how to define such a projection. Suppose $p \equiv a = b$, with $a \in Var_i$, $b \in Var_j$, and $i \neq j$. Introduce auxiliary variables x, y and consider the assertion

$$x = a \wedge y = b \wedge x = y$$

By definition, we have

$$proj_i(a = b) \equiv x = a \wedge x = y$$

$$proj_j(a = b) \equiv y = b \wedge x = y$$

Clearly, properties (1) and (2) are satisfied.

Definition 3.1 (assertion projection) *Let p be an assertion, $Var = \{x_1, \ldots, x_k\}$ be the set of processes variables, $Y = \{y_1, \ldots, y_k\}$ a set of variables, such that $Var \cap Y = \emptyset$. The projection of p over P_i, denoted $proj_i(p)$, is defined by:*

$$proj_i(p) = \bigwedge_{x_j \in Var_i} x_j = y_j \wedge trans(p)$$

avec $trans(p) = p[y_j/x_j],\ \forall j \in 1 \cdots k$

It can be easily shown that this definition satisfies properties (1) and (2) [Ada90].

3.2 A proof system

Like other proof systems designed for parallel context, this one is based on a compositional principle: local proofs are combined together into global proof upon synchronization points. Here, synchronization is expressed by *pulse sequentiality*: local proofs are made during a pulse, where the different assumptions relative to occurence of message emission are made; these proofs are combined together at the end of pulse, where the assumptions are solved, that is to say, impossible situations are dropped out. This resolution is similar to the one used in [Sou84] for CSP, where composition of local proofs take into account local communication histories which are mutually compared to keep only feaseable communications (pairs of emission-reception compatible with CSP's semantic); this technique is well suited to our synchronous model since local communication histories resume to zero or one message per channel, and can be reset at the beginning of each pulse (in some sense, this model behaves like a *Markovian system*, where the "past" can be forgotten).

Below, axioms and rules are given; two axioms (skip, assignment) and four rules (sequentiality, choice, loop, consequence) constitute the classical proof system for sequential programs with non-deterministic choice, as stated by Hoare in [Hoa69]. To this well-known system, one axiom and two rules have been added to deal with communication and synchronization. These are :

emission $\boxed{\{p[e/C_{ij}]\}\mathbf{P_j!e}\{p\}}$

This axiom is similar to assignment, namely $C_{ij} := e$

reception $\boxed{\dfrac{\{p\}\mathbf{S}\{q\}}{\{p[C_{ji}/x]\}[\mathbf{P_j?x} \rightarrow \mathbf{S}]\{(p[C_{ji}/x] \wedge C_{ji} = nil) \bigvee q\}}}$

This rule results from the assignment axiom and choice rule, namely:

$[\mathbf{C_{ji} \neq nil} \rightarrow \mathbf{x} := \mathbf{C_{ji}}; \mathbf{S} \square \mathbf{C_{ji}} = \mathbf{nil} \rightarrow \mathbf{skip}]$

pulse loop $\boxed{\dfrac{\begin{array}{c} i = 1, \ldots, n, \\ \{proj_i(p) \wedge b_i \wedge \bigwedge_{P_j \in V_i} C_{ij} = nil\}\mathbf{S_i}\{q_i\}, \\ \bigwedge_{i=1}^{n}(q_i \vee proj_i(p) \wedge \neg b_i \wedge \bigwedge_{P_j \in V_i}(C_{ij} = nil \wedge (C_{ji} = nil)) \Rightarrow p \end{array}}{\{p\}[\otimes[\mathbf{b_1} \rightarrow \mathbf{S_1}] \parallel \ldots \parallel \otimes[\mathbf{b_n} \rightarrow \mathbf{S_n}]]\{p \wedge \bigwedge_{i=1}^{n} \neg b_i\}}}$

Assertion p is a *loop invariant.* At the beginning of a pulse, the projection of p over a process is considered, channels are empty, guard of the pulse is either true or false. When guard is true, loop body is performed. When guard is false, loop body is void, ingoing and outgoing channels are empty. This rule is used for composition of local proofs.

We denote H the set of these nine axioms and rules. System H is small. Only one axiom (emission) and two rules (reception, pulse loop) are added to a classical system. Emission is an assignment, reception is an alternative involving an assignment and pulse loop is a generalization of classical sequential loop with an invariant. Moreover, local proofs can be set up separately on each process. A toy example given in the next section, shows how to use the pulse loop as a composition rule for local proofs.

Comparatively, the operationnal semantic is more intricate. This results from a deliberate choice, allowing a more concise proof system. In fact, semantic is but a necessary tool used to establish correctness and completness of the proof system, while H will be used as often as a synchronous program will have to be proved.

3.3 Correctness and completness

The proof system H is said to be *correct* (with regard to a given semantic) if every formula proven by H is correct in the given semantic :

$\vdash_H \{p\}\mathbf{S}\{q\} \Rightarrow \models \{p\}\mathbf{S}\{q\}$

where $\vdash_H \{p\}\mathbf{S}\{q\}$ means that $\{p\}\mathbf{S}\{q\}$ can be proved with H.

Correctness can be established by showing that each of the nine axioms and rules are correct in the semantic, e.g. for (7), (8), (9) :

emission $\models \{p[e/C_{ij}]\}P_j!e\{p\}$?

reception $\models (p \Rightarrow p1[C_{ji}/x] \wedge \{p1\}S\{q\}) \Rightarrow \models \{p\}[P_j?x \rightarrow S]\{p \wedge C_{ji} = nil \bigvee q\}$?

pulse loop

$$\begin{array}{c} \forall i = 1 \cdots n, \\ \models proj_i(p) \wedge b_i \wedge \bigwedge_{P_j \in V_i}(C_{ij} = nil) S_i \{q_i\} \\ \wedge \bigwedge_{i=1}^{n}(q_i \vee proj_i(p) \wedge \neg b_i \wedge \bigwedge_{P_j \in V_i}(C_{ij} = nil \vee C_{ji} = nil)) \Rightarrow p \\ \Downarrow \\ \models \{p\}[\otimes[b_1 \rightarrow S_1] \parallel \ldots \parallel \otimes[b_n \rightarrow S_n]]\{p \wedge \bigwedge_{i=1}^{n} \neg b_i\}? \end{array}$$

We don't give the detailed proofs here. They can be found in [Ada90]

Completness of proof system H means that every formula, correct in the semantic, can be proved with H. It is well known that no proof system is by itself complete. As an example, consider the formula $\{p[t/x]\}\mathbf{x} := \mathbf{t}\{true\}$. This formula is correct in the semantic, but cannot be proved with H. In fact, H cannot show $p \Rightarrow true$! The classical way to overcome this difficulty consists in using precondition and postcondition concepts, afterwards proving that each of the language constructions can be proved in H, e.g. :

emission $\vdash_H \{p\}P_j!t\{q\}?$

reception $\vdash_H \{p\}[P_j?x \rightarrow S]\{q\}?$

pulse loop $\vdash_H \{p\}[\otimes[b_1 \rightarrow S_1] \parallel \ldots \parallel \otimes[b_n \rightarrow S_n]]\{q\}?$

Interested reader will find in [Ada90] the detailed proofs of completness.

4 A toy illustrating example

The program is a simple producer-consumer scheme. Process P_1 produces a sequence of consecutive integers from 0 to *sup*. Only one number is produced during a pulse, and it is sent to a process P_2. The latter, upon receiving a number, rises it to square and terminates when it receives nothing during a pulse. The synchronous program is the following :

$P_1 :: i := 0; \otimes[i < sup \rightarrow i := i + 1; P_2!i]$
$P_2 :: j := 0; ch := true; \otimes[ch \rightarrow ch := false; [P_1?j \rightarrow j := j \times j; ch := true]]$

We illustrate the use of proof system H by proving that $I \equiv i \times i = j$ is a pulse invariant. To this end, the pulse loop rule must be used. First, each process is considered, then local assertions obtained at the end of the pulse are combined to make the final result. The projections of I over P_1 and P_2 are:

$trans(I) \equiv x \times x = y$
$proj_1(I) \equiv x = i \wedge x \times x = y$
$proj_2(I) \equiv y = j \wedge x \times x = y$

Assertions for P_1

$\{proj_1(I) \wedge i \leq sup \wedge C_{12} = nil\}$
$\{x = i \wedge x \times x = y \wedge i \leq sup \wedge C_{12} = nil\}$
$\{x + 1 = i + 1 \wedge x \times x = y \wedge i + 1 \leq sup + 1 \wedge C_{12} = nil\}$
$i := i + 1;$
$\{x + 1 = i \wedge x \times x = y \wedge i \leq sup + 1 \wedge C_{12} = nil\}$
$\{x + 1 = i \wedge x \times x = y \wedge i \leq sup + 1 \wedge i = i\}$
$P_2!i;$
$\{x + 1 = i \wedge x \times x = y \wedge i \leq sup + 1 \wedge C_{12} = i\}$

Assertions for P_2

$\{proj_2(I) \wedge ch \wedge C_{21} = nil\}$
$\{y = j \wedge x \times x = y \wedge ch \wedge C_{21} = nil\}$
$\{y = j \wedge x \times x = y \wedge \neg false\}$
$ch := false;$
$\{y = j \wedge x \times x = y \wedge \neg ch\}$
$[$
$\{C_{12} = C_{12} \wedge x \times x = y \wedge \neg ch\}$
$P_1?j \rightarrow$
$\{C_{12} = j \wedge x \times x = y \wedge \neg ch\}$
$\{C_{12} \times C_{12} = j \times j \wedge x \times x = y \wedge \neg ch\}$
$j := j \times j;$
$\{C_{12} \times C_{12} = j \wedge x \times x = y \wedge \neg ch\}$
$\{C_{12} \times C_{12} = j \wedge x \times x = y \wedge true\}$
$ch := true$
$\{C_{12} \times C_{12} = j \wedge x \times x = y \wedge ch\}$
$]$
$\{y = j \wedge x \times x = y \wedge \neg ch \wedge C_{12} = nil \vee C_{12} \times C_{12} = j \wedge x \times x = y \wedge ch\}$
$\{y = j \wedge x \times x = y \wedge C_{12} = nil \vee C_{12} \times C_{12} = j \wedge x \times x = y\}$

Combining assertions This combination is made by the conjonction of the two assertions

$x + 1 = i \wedge x \times x = y \wedge i \leq sup + 1 \wedge C_{12} = i$
$\vee x = i \wedge x \times x = y \wedge i > sup \wedge C_{12} = nil \wedge C_{21} = nil$

and

$y = j \wedge x \times x = y \wedge C_{12} = nil$
$\vee C_{12} \times C_{12} = j \wedge x \times x = y$
$\vee y = j \wedge x \times x = y \wedge C_{12} = nil \wedge C_{21} = nil \wedge \neg ch$

which can be simplified (since C_{21} and ch are not used in the proof) as $A \vee B$, $C \vee D$ where

$A \equiv \quad x + 1 = i \wedge x \times x = y \wedge i \leq sup + 1 \wedge C_{12} = i$
$B \equiv \quad x = i \wedge x \times x = y \wedge i > sup \wedge C_{12} = nil$
$C \equiv \quad C_{12} \times C_{12} = j \wedge x \times x = y$
$D \equiv \quad y = j \wedge x \times x = y \wedge C_{12} = nil$

Assertion A holds in the case where P_1 executes the pulse body (as computed above) and B in the case where P_1 has terminated. Similarly, C holds in the case where P_2 executes the pulse body and

receives a message, and D in the case where either P_2 has terminated or doesn't receive message during the pulse.

Now, a straightforward computation shows that

$A \wedge C \Rightarrow i \times i = j$

$A \wedge D \Rightarrow$ **false** (This case is can never occur in an execution)

$B \wedge C \Rightarrow$ **false** (This case can never occur in a computation)

$B \wedge D \Rightarrow i \times i = j$

Finally, we have proved

$\{i \times i = j\}$
$[\otimes[i < sup \rightarrow i := i + 1; P_2!i]$
$\|$
$\otimes[ch \rightarrow ch := false; [P_1?j \rightarrow j := j \times j; ch := true]]]$
$\{i \times i = j\}$

5 Derivation of a synchronous election algorithm

In this problem, each site has an identity, and all identities are distinct, so that we assume a total order on the set of site identities. Moreover, each site knows its own identity. To simplify the termination condition, we will make the additional assumption that every site knows the diameter d of the network. This assuption is not essential and could be removed, but more information should have to be exchanged and at the price of a more intricate derivation : our goal in this section is not to give the more general or efficient election algorithm, but to illustrate the synchronous distributed algorithms language and its proof system.

At the end of the algorithm, each site has to know :

- the identity of the site having a maximum identity (the elected site),
- its position within a spanning tree rooted at the elected site.

On each site i is located a process whose identity is P_i, the site identity. The two informations learnt by P_i during the algorithm will be implemented by variables max_i, $father_i$. In particular the site whose identity is $father_i$ defines a routing function from P_i to the elected site max_i, and all the sites will have the same value for max_i. The result can thus be expressed by the following predicate :

$R \equiv \bigwedge_{P_i \in V} (max_i = max(P_j | P_j \in V)$
$\wedge (father_i \in V_i \wedge d(max_i, P_i) = d(max_i, father_i) + 1) \vee (father_i = P_i \wedge max_i = P_i)))$

The algorithm will be based upon the following property. Let k ($1 \leq k \leq d$) be an integer and, for each site i, let $\mathcal{B}(i,k)$ be the subgraph spanned by sites at distance $\leq k$ from i (the ball of center i and radius k). Let $max_i(k)$, $father_i(k)$ be the solution to the problem on $\mathcal{B}(i,k)$, such as P_i sees it, that is to say:

$max_i(k) = \max(P_j | P_j \in \mathcal{B}(i,k))$

$(father_i \in V_i \wedge d(max_i(k), P_i) = d(max_i(k), father_i(k)) + 1)$
$\vee (father_i = P_i \wedge max_i(k) = P_i)$

Clearly, the following recurrence relation holds :

$$max_i(k+1) = \max\{max_i(k), \max(max_j(k) | P_j \in V_i)\}$$

$$father_i(k+1) = \begin{cases} father_i(k) & \text{if } max_i(k+1) = max_i(k) \\ father_{jmax}(k) & \text{if } max_i(k+1) = max_{jmax}(k) \text{ where } (P_{jmax} \in V_i) \end{cases}$$

Furthermore, $\forall i \ : \ max_i(0) = P_i \ , \ father_i(0) = P_i$.

From this recurrence relation follows that a process P_i learns $max_i(k+1), father_i(k+1)$ during a pulse, when itself and all its neighbours P_j know $max_j(k), father_j(k)$ at the end of the previous pulse.

Let, for each process P_i, denote by cpt_i the pulse counter. Consider the predicate

$$I \equiv I1 \wedge I2 \wedge I3 \equiv \bigwedge_{P_i \in V} I1_i \wedge I2_i \wedge I3_i \text{ where}$$

$$I1_i \equiv max_i = \max(P_j | d(P_j, P_i) \leq cpt_i)$$

$$I2_i \equiv \ ((father_i \in V_i \wedge d(max_i, P_i) = d(max_i, father_i) + 1) \vee \ (father_i = P_i \wedge max_i = P_i))$$

$$I3_i \equiv \bigwedge_{P_j \in V_i} cpt_i = cpt_j$$

and the initialization part

$\{true\}\ \mathbf{cpt_i := 0; max_i := P_i; father_i := P_i}\ \{INIT_i\}$ and thus :

$$INIT_i \equiv cpt_i = 0 \wedge max_i = P_i \wedge father_i = P_i$$

We have :

(1) $\bigwedge_{P_i \in V} INIT_i \quad \Rightarrow \quad I$ and
(2) $I \wedge \bigwedge_{P_i \in V} cpt_i = d \quad \Rightarrow \quad R$

whence the program skeleton follows:

$$\mathbf{cpt_i := 0; max_i := P_i; father_i := P_i; \otimes[cpt_i < d \rightarrow S_i]}$$

Now we have to find S_i and a predicate q_i such that I is a pulse loop invariant, that is, for each i:

$$\{proj_i(I) \wedge cpt_i < d \wedge \bigwedge_{P_j \in V_i} C_{ij} = nil\}\mathbf{S_i}\{q_i\} \text{ and}$$

$$\bigwedge_{i=1}^{n} q_i \vee \left(proj_i(I) \wedge cpt_i \geq d \wedge \bigwedge_{P_j \in V_i} (C_{ij} = nil \wedge C_{ji} = nil) \right) \Rightarrow I$$

Let's compute the projection of I over P_i:

$proj_i(I) \equiv x_i = cpt_i \wedge y_i = max_i \wedge z_i = father_i \wedge trans(I)$

$trans(I) \equiv I[x_i/cpt_i, y_i/max_i, z_i/father_i]$

The recurrence relation suggests the design of S_i: roughly, during a pulse each P_i must learn the values max_j from all its neighbours. To this end, at each pulse, each P_i must:

- during its *emission* phase send to each of its neighbours its current value max_i
- during its *reception* phase receive from its neighbours their value max_j, then update its own $max_i, father_i$.

Accordingly, we obtain:

$\{x_i = cpt_i \wedge y_i = max_i \wedge z_i = father_i \wedge cpt_i < d \wedge \bigwedge_{P_j \in V_i} C_{ij} = nil\}$
$[\nabla \mathbf{P_j \in V_i}\ \mathbf{P_j}!\mathbf{max_i}];$
$\{x_i = cpt_i \wedge y_i = max_i \wedge z_i = father_i \wedge cpt_i < d \wedge \bigwedge_{P_j \in V_i} C_{ij} = y_i\}$
$[\nabla \mathbf{P_j \in V_i}\ \mathbf{P_j}?\mathbf{m} \rightarrow \ [\mathbf{m} > \mathbf{max_i} \rightarrow \mathbf{max_i} := \mathbf{m}; \mathbf{father_i} := \mathbf{P_j}$
$\square$
$\mathbf{m} \leq \mathbf{max_i} \rightarrow \mathbf{skip}]$
$]$
$\{R_i \wedge (R1_i \vee R2_i)\}$ where
$R_i \equiv x_i = cpt_i \wedge trans(I) \wedge cpt_i < d \wedge \bigwedge_{P_j \in V_i} C_{ij} = y_i\}$
$R1_i \equiv max_i = y_i \wedge father_i = z_i \wedge \bigwedge_{P_j \in V_i} (max_i \geq C_{ji} \vee C_{ji} = nil)$
$R2_i \equiv \bigvee_{P_k \in V_i} (max_i = C_{ki} \wedge P_k = father_i \wedge C_{ki} > y_i \wedge \bigwedge_{P_j \in V_i} (max_i \geq C_{ji} \vee C_{ji} = nil)$

$R1_i$ holds when P_i received no value greater than max_i, whereas $R2_i$ holds when P_i received a value greater than the former max_i, kept by y_i.

$\{R_i \wedge (R1_i \vee R2_i)\}\ \mathbf{cpt_i} := \mathbf{cpt_i} + \mathbf{1}\ \{Q1_i \vee Q2_i\}$ where

$$Q_i \equiv x_i = cpt_i - 1 \wedge trans(I) \wedge cpt_i < d + 1 \wedge \bigwedge_{P_j \in V_i} C_{ij} = y_i\}$$

$Q1_i \equiv Q_i \wedge R1_i,\ Q2_i \equiv Q_i \wedge R2_i$

Now we have to show that

$$\bigwedge_{i=1}^{n} (Q1_i \vee Q2_i) \vee T_i \Rightarrow I \text{ where} T_i \equiv \left(proj_i(I) \wedge cpt_i \geq d \wedge \bigwedge_{P_j \in V_i} (C_{ij} = nil \wedge C_{ji} = nil) \right)$$

We can remark, first, that

$\forall P_i, P_j \in V\ :\ (Q1_i \vee Q2_i) \wedge T_j \Rightarrow false$

In fact,

$$\begin{aligned} Q1_i \vee Q2_i \wedge T_j &\Rightarrow cpt_i = x_i + 1 \wedge cpt_i < d+1 \wedge x_i = x_j \wedge cpt_j = x_j \wedge cpt_j \geq d \\ &\Rightarrow x_i + 1 < d+1 \wedge x_i = cpt_j \wedge cpt_j \geq d \\ &\Rightarrow cpt_j < d \wedge cpt_j \geq d \Rightarrow false \end{aligned}$$

Secondly, observe that, by projection definition:

$$\bigwedge_{P_i \in V} \left(proj_i(I) \wedge cpt_i \geq d \wedge \bigwedge_{P_j \in V_i} (C_{ij} = nil \wedge C_{ji} = nil) \right) \Rightarrow I$$

Thus it remains only to show $\bigwedge_{P_i \in V} (Q1_i \vee Q2_i) \Rightarrow I$

This can be split in two parts:

$$\bigwedge_{P_i \in V} (Q1_i \vee Q2_i) \equiv \bigwedge_{P_i \in V} \left((Q1_i \wedge \bigwedge_{P_j \in V_i} Q_i \right) \vee \bigwedge_{P_i \in V} \left(Q2_i \wedge \bigwedge_{P_j \in V_i} Q_i \right)$$

1) Proof of $Q1_i \wedge \bigwedge_{P_j \in V_i} Q_i \Rightarrow I1_i \wedge I2_i \wedge I3_i$

$$\begin{aligned} Q1_i \wedge \bigwedge_{P_j \in V_i} Q_i &\Rightarrow y_i = max_i \wedge y_i = \max(P_k | d(P_i, P_k) \leq x_i) \wedge x_i = cpt_i - 1 \\ &\quad \wedge \bigwedge_{P_j \in V_i} (max_i \geq C_{ji} \vee C_{ji} = nil) \\ &\quad \wedge \bigwedge_{P_j \in V_i} \left(x_i = x_j \wedge y_j = \max(P_\ell | d(P_i, P_\ell) \leq x_j) \wedge \bigwedge_{P_\ell \in V_j} C_{j\ell} = y_j \right) \\ &\Rightarrow max_i = \max(P_k | d(P_i, P_k) \leq cpt_i - 1) \wedge \bigwedge_{P_j \in V_i} max_i \geq y_j \\ &\quad \wedge \bigwedge_{P_j \in V_i} (cpt_i = cpt_j \wedge y_j = \max(P_\ell | d(P_j, P_\ell) \leq cpt_j - 1)) \\ &\Rightarrow max_i = \max(P_k | d(P_i, P_k) \leq cpt_i) \equiv I1_i \end{aligned}$$

$$\begin{aligned} Q1_i \wedge \bigwedge_{P_j \in V_i} Q_i &\Rightarrow z_i = father_i \wedge y_i = max_i \\ &\quad \wedge (father_i \in V_i \wedge d(y_i, P_i) = d(y_i, z_i) + 1) \vee (z_i = P_i \wedge y_i = P_i) \\ &\Rightarrow (father_i \in V_i \wedge d(max_i, P_i) = d(max_i, father_i) + 1) \vee \\ &\quad (father_i = P_i \wedge max_i = P_i) \equiv I2_i \end{aligned}$$

$$
\begin{aligned}
Q1_i \wedge \bigwedge_{P_j \in V_i} Q_i \quad &\Rightarrow \quad x_i = cpt_i - 1 \wedge \bigwedge_{P_j \in V_i} x_i = x_j \wedge x_j = cpt_j - 1 \\
&\Rightarrow \quad \bigwedge_{P_j \in V_i} cpt_i = cpt_j \equiv I3_i
\end{aligned}
$$

2) Proof of $Q2_i \wedge \bigwedge_{P_j \in V_i} Q_i \Rightarrow I1_i \wedge I2_i \wedge I3_i$

$$
\begin{aligned}
Q2_i \wedge \bigwedge_{P_j \in V_i} Q_i \quad &\Rightarrow \quad max_i = C_{ki} \wedge C_{ki} > y_i \wedge P_k \in V_i \\
&\qquad \wedge y_i = \max(P_k | d(P_i, P_k) \leq x_i) \wedge x_i = cpt_i - 1 \\
&\qquad \wedge \bigwedge_{P_j \in V_i} (max_i \geq C_{ji} \vee C_{ji} = nil) \\
&\qquad \wedge \bigwedge_{P_j \in V_i} (x_j = x_i \wedge y_j = \max(P_\ell | d(P_j, P_\ell) \leq x_j) \\
&\qquad \wedge \bigwedge_{P_\ell \in V_j} C_{j\ell} = y_j \\
&\Rightarrow \quad max_i = y_k \wedge y_k = \max(P_\ell | d(P_k, P_\ell) \leq cpt_i - 1) \wedge P_k \in V_i \\
&\qquad \wedge \bigwedge_{P_j \in V_i} (cpt_i = cpt_j \wedge max_i > y_j \wedge y_j = \max(P_\ell | d(P_j, P_\ell) \leq cpt_j - 1) \\
&\Rightarrow \quad max_i = \max(P_k | d(P_i, P_k) \leq cpt_i) \equiv I1_i
\end{aligned}
$$

$$
\begin{aligned}
Q2_i \wedge \bigwedge_{P_j \in V_i} Q_i \quad &\Rightarrow \quad father_i = P_k \wedge max_i = C_{ki} \wedge C_{ki} > y_i \wedge P_k \in V_i \\
&\qquad \wedge y_i = \max(P_\ell | d(P_i, P_\ell) \leq x_i) \wedge x_i = cpt_i - 1 \\
&\qquad \wedge \bigwedge_{P_j \in V_i} (max_i \geq C_{ji} \vee C_{ji} = nil) \\
&\qquad \wedge \bigwedge_{P_j \in V_i} (x_j = x_i \wedge y_j = \max(P_\ell | d(P_j, P_\ell) \leq x_j) \\
&\qquad \wedge \bigwedge_{P_\ell \in V_j} C_{jl} = y_j \\
&\Rightarrow \quad P_k \in V_i \wedge father_i = P_k \wedge max_i > \max(P_\ell | d(P_i, P_\ell) < cpt_i) \\
&\qquad \wedge max_i = \max(P_\ell | d(P_k, P_\ell) < cpt_i) \\
&\Rightarrow \quad father_i \in V_i \wedge father_i = P_k \\
&\qquad \wedge d(P_i, max_i) \geq cpt_i - 1 \wedge d(P_k, max_i) \leq cpt_i - 1
\end{aligned}
$$

From distance inequality: $d(P_i, max_i) \leq d(P_i, P_k) + d(P_k, max_i) \leq cpt_i$ whence, on the one hand,

$d(P_i, max_i) = cpt_i$ and on the other hand

$d(P_k, max_i) = cpt_i - 1$. From these two relations follows

$d(max_i, P_i) = d(max_i, father_i) + 1 \Rightarrow I2_i$

$Q2_i \wedge \bigwedge_{P_j \in V_i} Q_i \Rightarrow I3_i$ is obtained like in the first case.

The proof is complete, and the text of the program follows:

$\mathbf{P_i}$ ::$\mathbf{cpt_i := 0;\ max_i := P_i;\ father_i := P_i}$
$\otimes[\mathbf{cpt_i <> d} \rightarrow [\nabla \mathbf{P_j \in V_i}\ \mathbf{P_j}!\mathbf{max_i}];$
$[\nabla \mathbf{P_j \in V_i}\ \mathbf{P_j}?\mathbf{m} \rightarrow$
$[\mathbf{m > max_i} \rightarrow \mathbf{max_i := m;\ father_i := P_j}$
$\square$
$\mathbf{m \leq max_i} \rightarrow \mathbf{skip}]$
$];$
$\mathbf{cpt_i := cpt_i + 1}$
]

Note that this rough algorithm is not message optimal in the communication context (arbitrary topology, global knowledge of diameter, synchronous communications): a process needs to send its current value max_i only if this value has been improved during the previous pulse. Also, a process doesn't need to send its current value to proceses who know it already, namely those who sent him this value during previous pulses.

6 Conclusion

Design and understanding of distributed algorithms requires a global view of the set of processes. This requirement is seldom met, due to the great number of possible behaviors in a distributed environment. One way to achieve it consists in reducing processes asynchronism. A synchronous distributed algorithm will thus be designed as a sequence of steps separated by synchronization points. This conception gave raise to a proof system suited to such synchronous distributed algorithms.

In this paper the design of a synchronous distributed algorithm solving the election problem was obtained through a derivation method based upon the proof system. The proof was brought to completion, nevertheless the lack of proof assistance tools is sorely felt. On the other hand, finding an invariant from the problem specification cannot yet be carried out automatically: this is the part of imagination left to the designer.

The fact nonetheless remains that such a proof system is a significant step on the way to the conception of a programming environment offered to distributed application designers. Another improvement would consist of a compiler allowing programs, expressed in a synchronous language similar to ours, to be implemented on real distributed memory machines like hypercubes or transputer networks. Such a software tool would also be in charge of producing and mixing the part of code due to the synchronizer layer: a good basis for this could be ECHIDNA in its version including superimposition [Cai89].

References

[Ada90] M. Adam. Synchronisme des systèmes distribués. Thèse, Univ. Rennes I, Rennes, Mai 1990.

[AFP80] K.R. Apt, N. Francez, and De Roever W. P. A proof system for communicating sequential processes. *ACM Transactions on programming languages and systems*, 2(3):359–385, July 1980.

[AIR88] M. Adam, Ph. Ingels, and M. Raynal. The meaning of synchronous distributed algorithms run on asynchronous distributed systems. In *The Third International Symposium on Computer and Information Sciences, Izmir*, pages 307–316, November 1988.

[Awe85] B. Awerbuch. Complexity of network synchronization. *Journal of ACM*, 32(4):801–823, October 1985.

[Cai89] B. Caillaud. *The superimposition of Estelle programs: A tool for the implementation of observation and control algorithms.* Rapport de Recherche 1102, INRIA, October 1989.

[CCGZ87] C.T. Chou, I. Cidon, I.S. Gopal, and S. Zaks. Synchronizing asynchronous bounded delay networks. In *Proc. 2nd. Int. Workshop on Distributed Algorithms*, Amsterdam, July 1987. Springer Verlag LNCS 312 (1988).

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 63–75, February 1985.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Fid88] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proc.* 11^{th} *Australian Computer Science Conference*, pages 55–66, 1988.

[FLS87] A. Fekete, N. Lynch, and L. Shrira. A modular proof of correctness for network synchroniser. In *Proc. 2d Int. Workshop on Distributed Algorithms*, Amsterdam, July 1987. Springer Verlag LNCS 312 (1988).

[Gri81] D. Gries. *The science of programming.* Springer-Verlag, 1981.

[Hen79] G.D. Hennessy, M.C.B. amd Plotkin. Full abstraction for a simple parallel programming language. In *Proceedings of 8th Symposuim of Foundations of Computer Sciences, Lecture Notes in Computer Sciences, Springer-Verlag*, 1979.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12(10):576–580,583, October 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, August 1978.

[HPR89] J.M. Hélary, N. Plouzeau, and M. Raynal. A characterization of a particular class of distributed snapshots. In *Proc. International Conference on Computing and Information (ICCI'89), Toronto*, North-Holland, may 23–27 1989.

[HR88] J.-M. Hélary and M. Raynal. *Synchronisation et contrôle des systèmes et des programmes répartis.* Eyrolles, Septembre 1988. English translation to appear, Wiley, 1990.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications. of the ACM*, 21(7):558–565, July 1978.

[LY87] T.H. Lai and T.H Yang. On distributed snapshots. *Inf. Proc. Letters*, 25:153–158, 1987.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms, Bonas, France*, North Holland, September 1988.

[MO83] K. Marzullo and S. Owiki. Maintaining time in a distributed system. In *ACM Operating Systems Rev.*, pages 44–54, 1983.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[PU87] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. In *6th Annual ACM Symposium on Principles of Distributed Computing*, pages 77–85, August 1987.

[Sou84] N. Soundarajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on programming languages and systems*, 6(4):647–662, October 1984.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 530 **SEMI-GRANULES AND SCHIELDING FOR OFF-LINE SCHEDULING**
Bernard LE GOFF, Paul LE GUERNIC, Julian ARAOZ DURAND
Avril 1990, 46 Pages.

PI 531 **DATA-FLOW TO VON NEUMANN : THE SIGNAL APPROACH**
Paul LE GUERNIC, Thierry GAUTIER
Avril 1990, 22 Pages.

PI 532 **OPERATIONAL SEMANTICS OF A DISTRIBUTED OBJECT-ORIENTED LANGUAGE AND ITS Z FORMAL SPECIFICATION**
Marc BENVENISTE
Avril 1990, 100 Pages.

PI 533 **ADAPTATION DE LA METHODE DE DAVIDSON A LA RESOLUTION DE SYSTEMES LINEAIRES : IMPLEMENTATION D'UNE VERSION PAR BLOCS SUR UN MULTIPROCESSEUR**
Miloud SADKANE, Brigitte VITAL
Avril 1990, 34 Pages.

PI 534 **DIFFUSE INTERREFLECTIONS. TECHNIQUES FOR FORM-FACTOR COMPUTATION**
Xavier PUEYO
Mai 1990, 28 Pages.

PI 535 **A NOTE ON GUARDED RECURSION**
Eric BADOUEL, Philippe DARONDEAU
Mai 1990, 10 Pages.

PI 536 **TOWARDS DOCUMENT ENGINEERING**
Vincent QUINT, Marc NANARD, Jacques ANDRE
Mai 1990, 20 Pages.

PI 537 **YALTA : YET ANOTHER LANGUAGE FOR TELEOPERATE APPLICATIONS**
Jean-Christophe PAOLETTI, Lionel MARCE
Juin 1990, 32 Pages.

PI 538 **SYNCHRONOUS DISTRIBUTED ALGORITHMS : APROOF SYSTEM**
Michel ADAM, Jean-Michel HELARY
Juin 1990, 20 Pages.

PI 539 **CONCEPTION DE DESCRIPTEURS GLOBAUX EN ANALYSE DU MOUVEMENT A PARTIR D'UN CHAMP DENSE DE VECTEURS VITESSES APPARENTES**
Henri NICOLAS, Claude LABIT
Juin 1990, 38 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249 - 6399